LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-676454

Silo/HDF5 and Portable, Scalable, Parallel I/O

M. C. Miller

August 20, 2015

Silo/HDF5 and Portable, Scalable, Parallel I/O
Gaithersberg, DC, United States
September 15, 2015 through September 17, 2015

Disclaimer

Silo/HDF5 and Portable, Scalable, Parallel I/O

Mark C. Miller, miller86@llnl.gov

How many developers are involved and how is the development structured?

Silo is a BSD Open Source library for I/O of scientific computing data to portable, binary disk files. Development began in 1993 in B Division at LLNL. By 1998, Silo expertise helped to bring about HDF5 and then added HDF5 as a middleware layer within Silo. Throughout its life, 15-20 different developers have contributed a total of about 5-7 man-years. Vendor contracts for HDF5 enhancements to support Silo have amounted to ~$2M. Occasionally Silo gets community contributions as patches to a release. Currently Silo has one primary developer at 0.2 FTE plus an occasional Windows developer both located at LLNL. Silo consists of 210K lines of C code. Why C? It minimizes issues supporting C, C++ and Fortran callers with a common implementation. Silo is hosted with a Subversion repo, web site for releases and email list at LLNL and a Redmine site at ORNL. Silo's user base spans many DOE (e.g. LLNL, ANL, NERSC, ORNL), DoD, academic (e.g. TACC) and commercial sites. Interest in Silo continues to grow due to its use in flagship LLNL codes like Ale3d and VisIt and, more recently, has been experiencing a resurgence in Fortran usage.

Primary methods

Silo was developed to address a fundamental software engineering challenge; to foster the development of portable, reusable software through a common API, data model and file format for storing and exchanging data. Raw performance, although important, has proven to be a secondary consideration to common, reusable software Silo enables. Although Silo is a serial I/O library, key feature enable its use in scalable, parallel applications using the Multiple Independent File (MIF) I/O paradigm. In MIF-IO, a mesh is decomposed into N_D domains, processed on N_T tasks and stored in N_F files where N_D, N_T, N_F can be chosen entirely independently. Given N_D=60, a Silo application can run 1:1 domains-to-tasks with N_T=60, N_F=6, 2:1 with N_T=30, N_F=3 files, or even 3:1 with N_{T1}=12, 4:1 with N_{T2}=6 of N_{T1}+N_{T2}=18 total tasks with N_F=8 files. MIF-IO has many advantages; the programming model is simple; its easy to retrofit existing sequential apps; there is no required global-to-local and local-to-global remapping during I/O; compression and other data reduction services are easily handled even in parallel; good performance demands very little from the underlying file system; application controlled throttling of I/O resources is easy; great flexibility is permitted in the allocation of compute resources for any given problem setup. Finally, MIF-IO is completely analogous to how "big data" Map/Reduce I/O is handled as *shards* in the data sciences community. HDF5 provides a key layer of abstraction in the HPC I/O *stack*. The I/O Stack is a layering of software abstractions entirely analogous to IP Protocol Stack. Using HDF5 as a middleware layer, we can often address performance issues by adjustments to HDF5 or Silo without touching applications; compression to improve I/O and storage efficiency, application-level checksumming to mitigate file system reliability issues; a BG/P specific Virtual File Driver to gain 50x performance improvement to name just a few.

Types of problems/domains/science application problems

Due to its generally useful mesh and field abstractions, Silo is used in many different application domains. At LLNL, it is used for RadHydro, FEM, CFD, CEM, MD, and Structural Mechanics codes to name a few.

Scale of resources commonly used for production runs

Silo is routinely used at scales ranging from dual-core laptops to 10^5+ core LCF computing platforms at various DOE sites. Its use in runs of 10^4+ cores is typical. While Windows/OSX laptops can hardly be considered scalable, the ability to develop, test and debug Silo applications on such systems is nonetheless an invaluable enablement to developing and supporting capabilities for extreme scale.

Supercomputers regularly used

Silo sees use predominantly at LLNL but also at ANL, ORNL and other DOE and DoD sites, academic institutions such as TACC and and even some commercial companies.

Libraries/tools for prototyping

Valgrind, Totalview, HDF5, GNU compiler tools (gcov, mudflap)

Prepared by LLNL under Contract DE-AC52-07NA27344. LLNL-CONF-676454

Describe efforts to develop code portable across diverse architectures

The HPC I/O Stack offers a number of layers at which portability solutions have been developed. In the late 90's the MPI-IO interface provided portability across GPFS, Lustre and PVFS file systems being developed in that era. The HDF5 interface provides portability of data across different CPU architectures (e.g. endian-ness, float-format), compression schemes (e.g. gzip, szip, fpzip, hzip) and portability of interface across different storage system interfaces (MPI-IO, Posix, Mmap, Globus, NDGM, VOL). The Silo layer has provided portable mesh and field abstractions across different I/O libraries (HDF5, netCDF, PDB, Tarus) though HDF5 is now the only of these interfaces routinely used in scalable applications using Silo.

Where were the abstractions?

The HPC I/O stack is a set of layered abstractions. At the top, applications use the Silo API and its computational object abstractions (meshes and fields). In turn, Silo's mesh and field abstractions are implemented in terms of programming language abstractions (arrays, structs, lists). These language abstractions are in turn implemented in terms secondary storage interfaces and abstractions (e.g. MPI-IO, stdio, files, byte lengths and offsets). This multi-layer I/O stack has provided great flexibility in incorporating performant solutions. In general, the best solutions have been implemented as deeply within the I/O stack as possible leaving the upper level Silo API and codes that use it untouched.

How much code re-use was possible? If something was not possible, describe why.

Throughout its 20+ year life and through several transitions in order of magnitude of scale, a majority of the Silo library has proven reusable for I/O needs of scalable HPC codes. On a few occasions, new objects that fit within the current abstractions have been added. But, the original abstractions have continued to work. There is, however, a growing need to support high order elements introducing a fundamental change in Silo's mesh abstractions. In one instance, an object was added to Silo which suffered from scaling issues and was replaced. Although Silo needs some housekeeping, refactoring and modernization, its API and data model remain applicable to current scalable applications. The fact that the same Silo API has supported codes for many years has meant that little effort/cost has been necessary to re-engineer the I/O portions of these codes as scales have grown.

What successes have you had with performant code across different architectures? Were the same algorithms applicable at all across the architectures?

The basic MIF parallel I/O paradigm has served well through several transitions in order of magnitude of scale. On the other hand, a conditionally compiled MPI-IO optimization code block in HDF5 had been accidentally disabled and gone undetected in multiple releases of HDF5. This experience underscores the fact that manually maintaining multiple implementations of a capability can be a challenge.

What approaches did you reject and why? What was the leading contender rejected?

Where Silo is concerned, we have rejected concurrent I/O to a single, shared file (SSF) as a basic I/O paradigm due to its inflexibility and challenges in tuning it to get good performance. However, this decision is worth revisiting periodically to assess if logistical constraints and performance tuning have improved.

What is your greatest fear going to exascale for application portability and functionality?

Where I/O is concerned, we need to transition from *push* to *pull* paradigms. And, we need the added ability for users to control *when* and *how much* data is pulled. In a pull paradigm, applications simply announce they hayve *data ready* at the end of each main compute cycle. A *restart* consumer, for example, monitors system MTBF params, power consumption, etc. and pulls data from applications as needed to minimize restart costs. Likewise, post-processing and analysis consumers would pull data from codes to meet the needs of user's analysis workflows. Pulling data would include the ability to reduce the data being captured by various means including deciding when data is pulled by various *triggering conditions* and how much data is pulled by various data reductions including spatio-temporal & feature-based subsetting, resolution reduction, precision reduction, statistical reductions, lossless and loss-controlled compression. In this new pull paradigm for I/O, the code to manage data capture for *any* purpose is moved out of apps (where it is currently often duplicated among apps) and centralized to a common scalable data management system that all codes can benefit from.

Prepared by LLNL under Contract DE-AC52-07NA27344. LLNL-CONF-676454